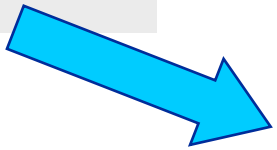
Assignment 5 – Task 1
Recurrence elimination

- Observation: recurrence is only “pseudo” as array contents can be computed from i directly:

```
for(i=1;i<N;i++) {
  b[i]=1+i;
  c[i]=b[i-1]+i;
  d[i]=c[i-1]+i;
}
```



```
b[1]=2
c[1]=b[0]+1
d[1]=c[0]+1
b[2]=3;
c[2]=b[1]+2;
d[2]=c[1]+2;
#pragma omp parallel for
for(i=3;i<N;i++) {
  b[i]=1+i;
  c[i]=2*i;
  d[i]=3*i-2;
}
```

Assignment 5 – Task 2

- Pi calculation using a Monte Carlo method and OpenMP

```
cout << "No. of threads: " << numthreads=omp_get_max_threads() << endl;
double rcp = 1./RAND_MAX;
long sum=0;
```

```
#pragma omp parallel reduction(+:sum)
{
   int myID = omp_get_thread_num();
   unsigned int seed = 6*myID;
   double x,y;

#pragma omp for
   for (int i = 0; i < nIterations; i++) {
      x = (rand_r(&seed)*rcp); y = (rand_r(&seed)*rcp);
      if (x*x + y*y <= 1.0) ++sum;
   }

}
```

```
double pi = (4.0 * sum) / nIterations;
```

- Performance / accuracy results
  - 10 threads @ 1 sec runtime (2.2 GHz), iterations = 6.2×10^8: relative error = 2.3×10^{-6}
  - Perfect scaling 1→10 threads

Assignment 5 – Task 2

- **Common errors**
  - `omp_get_thread_num()` and `omp_get_num_threads()` are only useful inside parallel regions
    - ```
      #pragma omp parallel
      {
        if(omp_get_thread_num()==0)
          numthreads=omp_get_num_threads();
      }
      ```
  - Forgot to privatize random seeds → no additional statistics, strange random number sequences
  - Put random seeds into small array:
    `x = (rand_r(&seeds[myID])*rcp);`
    → massive false sharing leads to extremely bad scalability
  - `rand()` works correctly, but very slowly (internal lock on the seed?)
  - Watch your data types – when handling large counts, using `long` may be mandatory
      ```

Assignment 5 – Task 2

- General advice
  - Take care with random numbers; use the available range as far as possible
  - Random seeds should not be too close together (empirical experience)
    - Factor of 2 in accuracy for choosing **`seed=6*myID`** instead of **`myID`** alone
    - Correct way: use parallel RNG, by fast-forwarding seeds
  - It is easier to make thread-local automatic variables inside the parallel region than to privatize globals
  - Keep “expensive” divide and sqrt operations out of inner loop

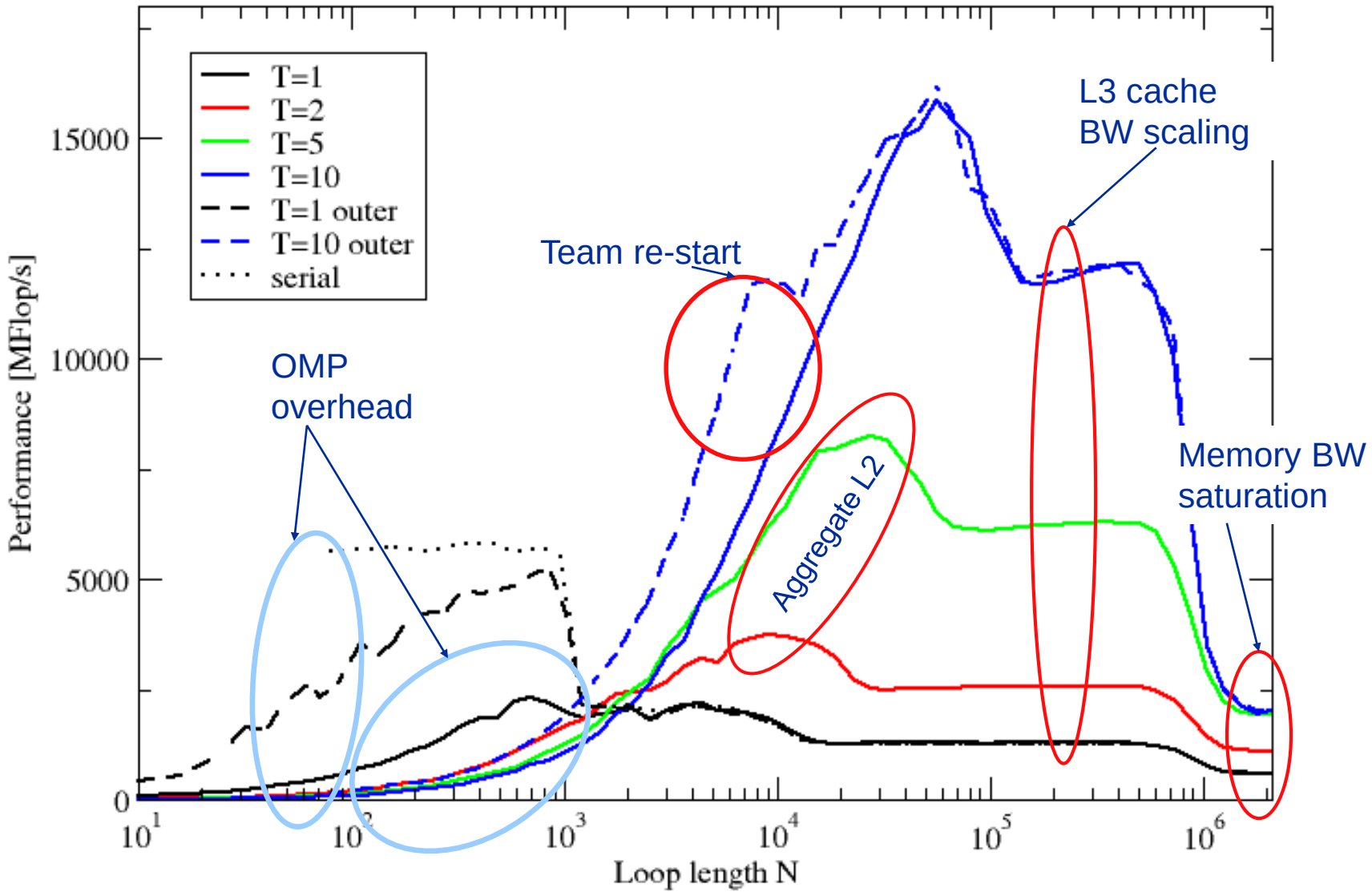
Assignment 5 –Task 3: Vector triad on Emmy

```
double precision, dimension(:), allocatable :: A,B,C,D

allocate(A(1:N),B(1:N),C(1:N),D(1:N))
A=1.d0; B=A; C=A; D=A
!$OMP PARALLEL private(j)
do j=1,NITER
!$OMP PARALLEL DO
  do i=1,N
    A(i) = B(i) + C(i) * D(i)
  enddo
!$OMP END PARALLEL DO
  if(.something.that.is.never.true.) then
    call dummy(A,B,C,D)
  endif
enddo
!$OMP END PARALLEL
```

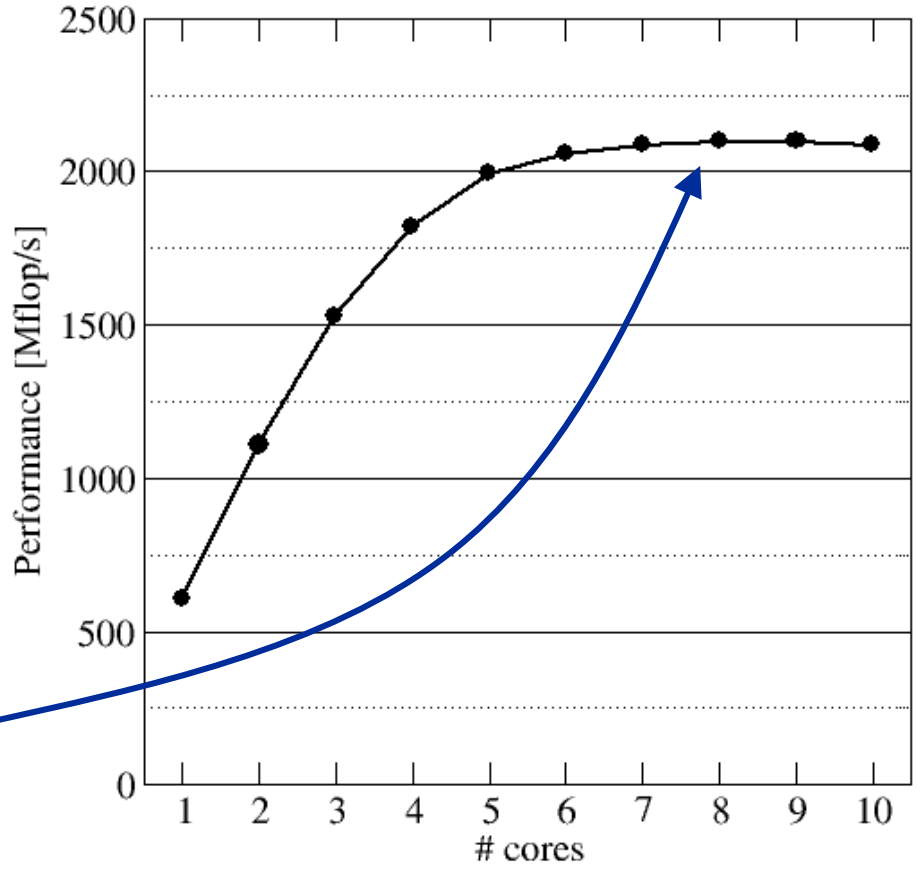
Outer parallel

Assignment 5 –Task 3:
Vector triad on Emmy (1 socket @ 2.2 GHz)



Assignment 5 –Task 3: Vector triad on Emmy (1 socket @ 2.2 GHz)

- Observations
  - OpenMP with 1 thread is significantly slower than serial code at small sizes
  - L1: No scalability at all, performance even goes down as #cores goes up when N < 2000
  - L2: Just large enough to show some scalability
  - L3: Scales fine although it's a shared resource
  - Memory: saturation pattern
- Memory BW at saturation
  - $b = 2100 \frac{Mflop}{s} \times 20 \frac{B}{F} = 42\ GB/s$



Assignment 5 – Task 4

Dense triangular MVM

```
#pragma omp parallel private(i,k)
{
  for(k=0; k<niter; k++){
#pragma omp for schedule(runtime)
    for(j=0; j<size; ++j)
      for(i=0; i<=j; ++i)
        c[j]=c[j]+a[i+size*j]*b[i];
    if(c[size >> 1]<0.0) whatever();
  }
}
```

$B_C^{mem} = 4\ \mathrm{B/F}$ (DP)

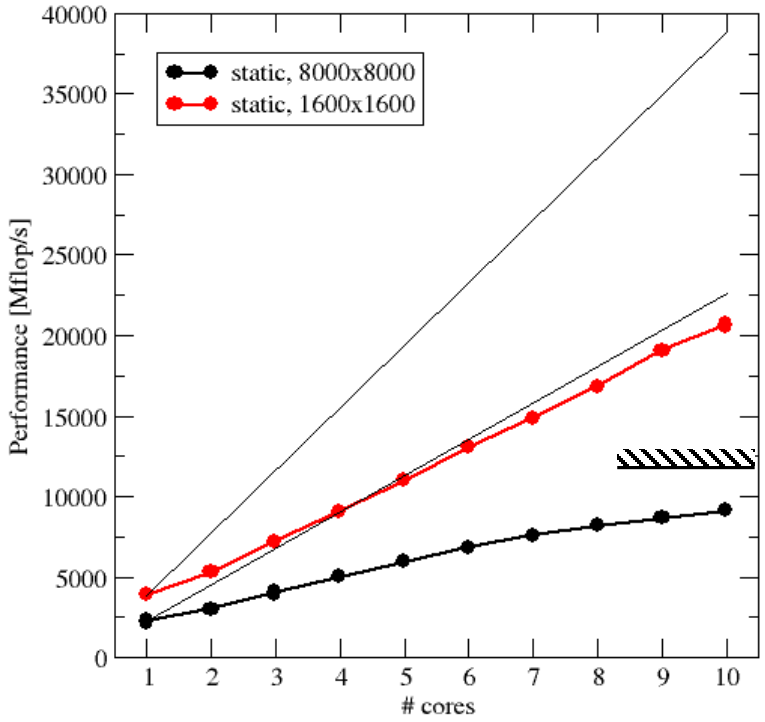
$b_S = 47\,\frac{GB}{s}$ (read-only)

$\rightarrow P = \frac{b_S}{B_c} = \frac{47\frac{GB}{s}}{4\frac{B}{F}} = 11.8\,\frac{GF}{s}$

- Parallelize outer loop (least overhead)
- Several overlapping effects:
  - Load imbalance
  - Bandwidth saturation (this is where Roofline applies!)
  - Prefetcher madness
  - OpenMP overhead (at small sizes)

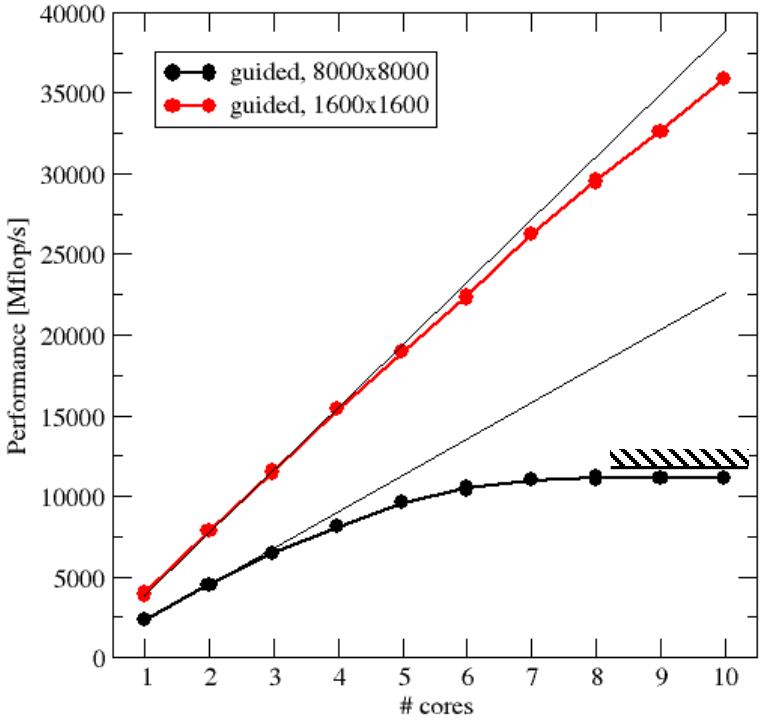
Assignment 5 – Task 4

static

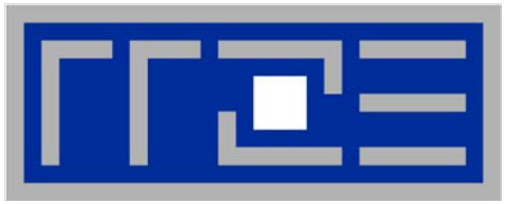


- Expected saturation pattern @ large size not really visible
- Expect good scaling @ smaller size, but weird scaling pattern because of load imbalance

guided,1



- Large size now shows saturation in accordance with Roofline model
- Smaller size shows very good scaling (L3 cache is scalable resource)
- → proper scheduling lets us reach the hardware limits



Simultaneous multithreading (SMT) (a.k.a. Hyper-Threading)

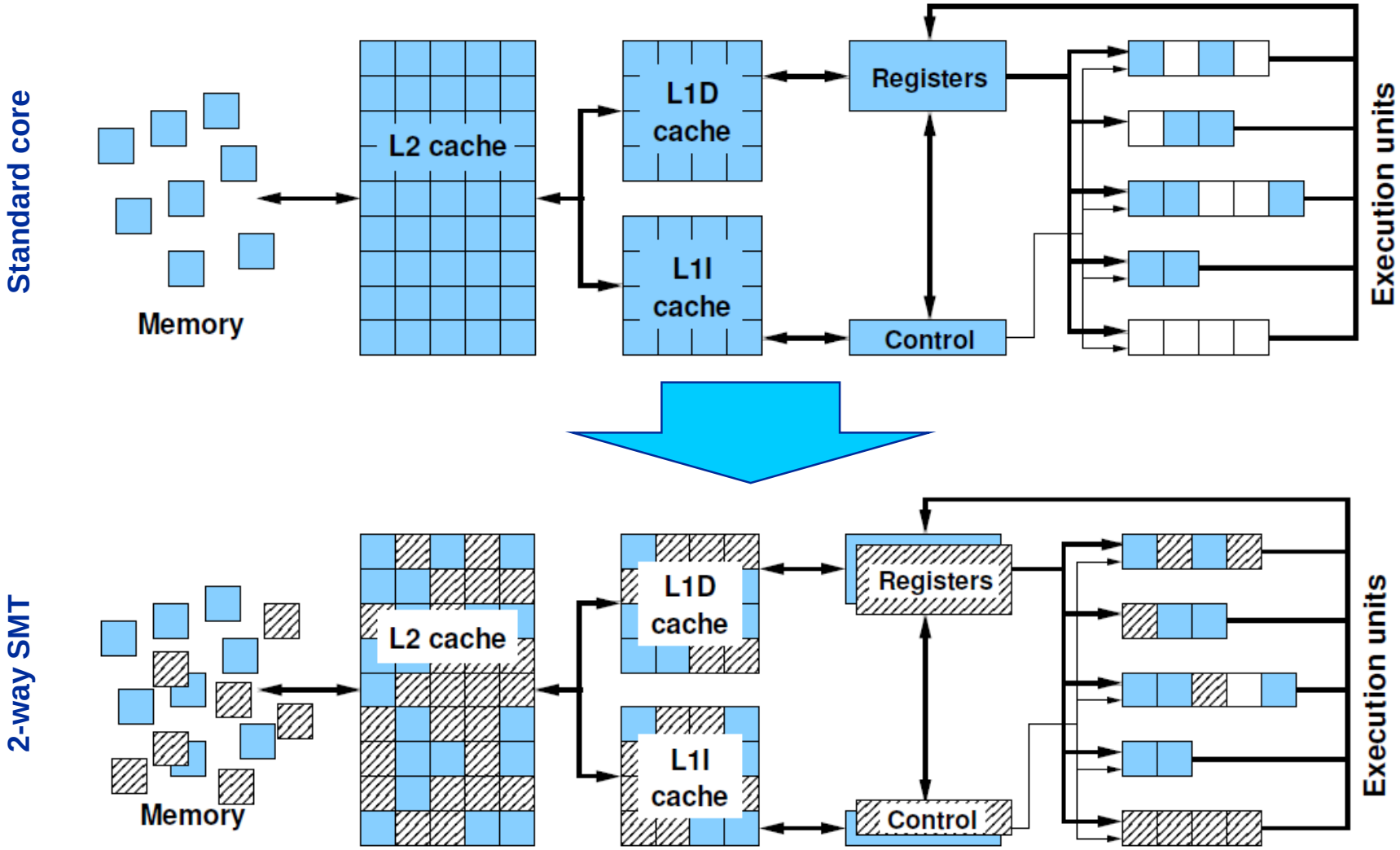
Principles and performance impact

SMT vs. independent instruction streams

Facts and fiction

SMT Makes a single physical core appear as two or more "logical" cores → multiple threads/processes run concurrently

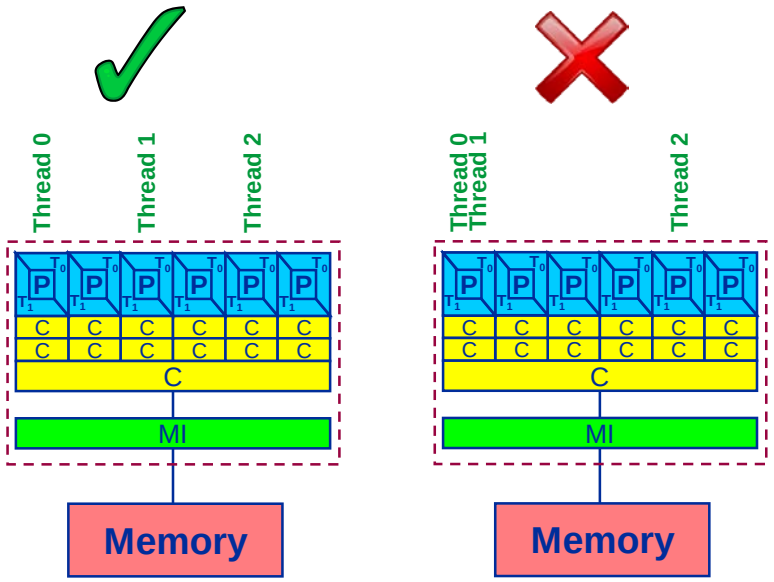
- SMT principle (2-way example):



SMT impact

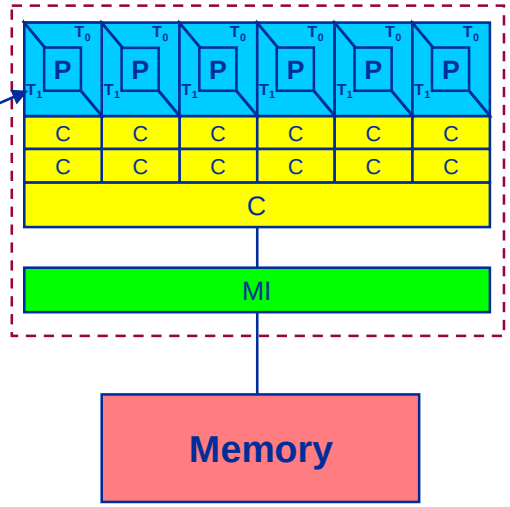
- SMT is suited for increasing the **in-core** throughput
  - With multiple threads/processes running concurrently
- Scientific codes tend to utilize chip resources quite well
  - Standard optimizations (loop fusion, blocking, …)
  - High data and instruction-level parallelism
  - Exceptions do exist

- SMT is an important topology issue
  - SMT threads share almost all core resources
    - Pipelines, caches, data paths
  - Affinity matters!
  - If SMT is not needed
    - pin threads to physical cores
    - or switch it off via BIOS etc.



SMT nomenclature

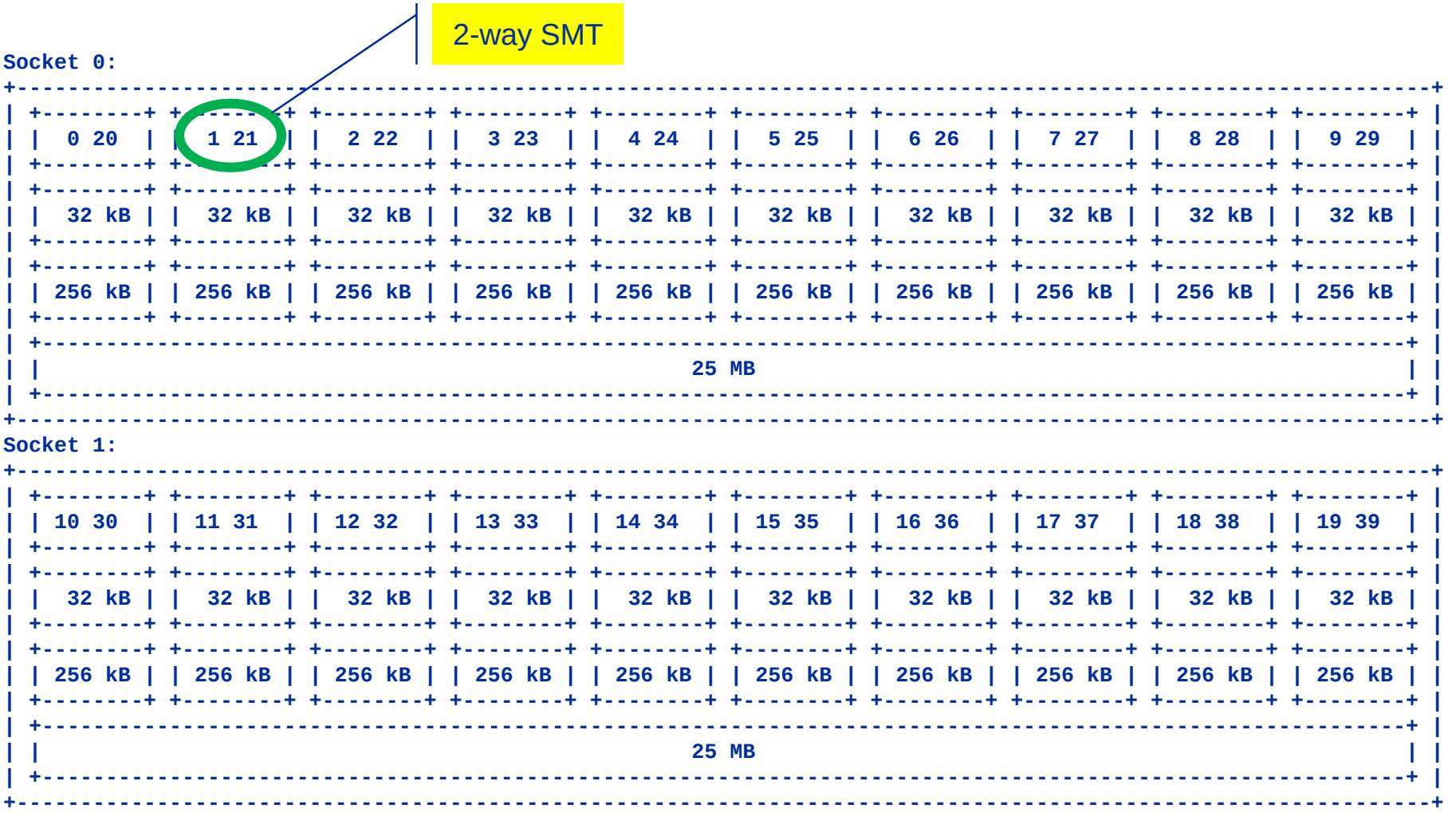
- There is a lot of confusion due to historical reasons
- “Thread” usually means “software thread”
  - In the sense of OpenMP, POSIX threads, etc.
- An “SMT thread” is a hardware feature
  - Also called “hardware thread” or “hyper-thread”



- **A software thread always runs on exactly one SMT thread**
  - Which SMT thread it runs on is determined by affinity settings
- In practice, the distinction is often not made consistently, leading to confusion

SMT and pinning with LIKWID

Likwid-topology output (Emmy compute node): numbers are hardware thread IDs



Examples for “logical” pinning

- Run 20 OpenMP threads on first socket with SMT

```
$ likwid-pin -c S0:0-19 ./a.out
```

Caveat: likwid-pin numbering is in this case “physical cores first,” i.e., the mapping is “round-robin style”:

OMP thread	HW thread
0	0
1	1
…	…
9	9
10	20
11	21
…	…

How to ignore the SMT threads and use 20 physical cores only?

```
$ likwid-pin -c N:0-19 ./a.out
```

The “N” prefix uses physical-first logical numbering across the whole node
Alternative (same effect): **S0:0-9@S1:0-9**

SMT and pinning with LIKWID

- More control via the “expression-based interface,” which employs “compact” logical numbering through the hyper-threads:

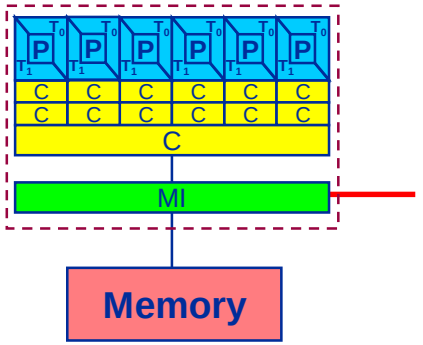
```
$ likwid-pin -c E:S0:20:1:1 ./a.out
```

The means: 20 threads, in chunks of 1, with a stride of 1

OMP thread	HW thread
0	0
1	20
2	1
3	21
…	…
18	9
19	29

SMT impact

- Caveat: SMT threads share all caches!



- Possible benefit: Better pipeline throughput
  - Filling otherwise unused pipelines
  - Filling pipeline bubbles with other thread's executing instructions:

Thread 0:

```
do i=1,N
  a(i) = a(i-1)*c
enddo
```

Dependency → pipeline stalls until previous MULT is over

Thread 1:

```
do i=1,N
  b(i) = s*b(i-2)+d
enddo
```

Unrelated work in other thread can fill the pipeline bubbles

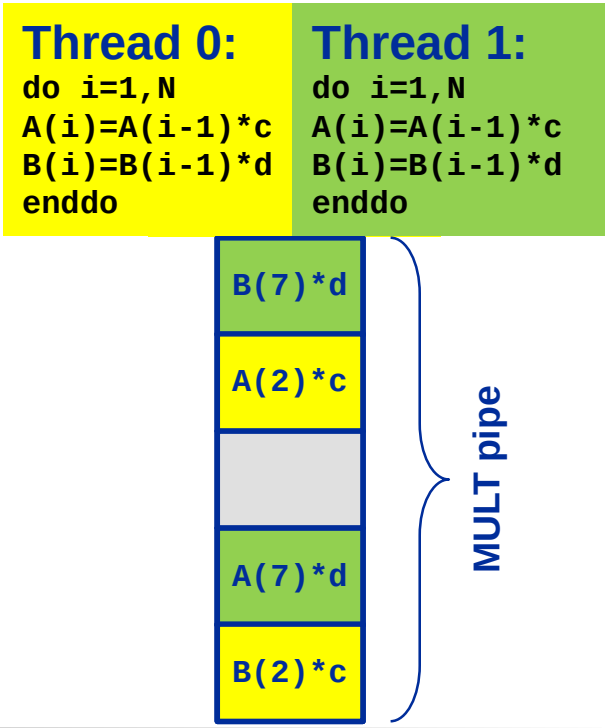
  - Beware: Executing it all in a single thread (if possible) may reach the same goal without SMT:

```
do i=1,N
  a(i) = a(i-1)*c
  b(i) = s*b(i-2)+d
enddo
```

The ideal workload for SMT

Simple loop-carried dependency benchmark `A(i) = s*A(i-1)`

- Bottleneck: MULT pipeline latency
  - Haswell CPU: 5 cy/it best case
- Running 2 threads via SMT: expect 2.5 cy/it if no other bottlenecks turn up
- Further improvement?
  - Multiple independent streams of instructions per thread
- What about the data transfer?

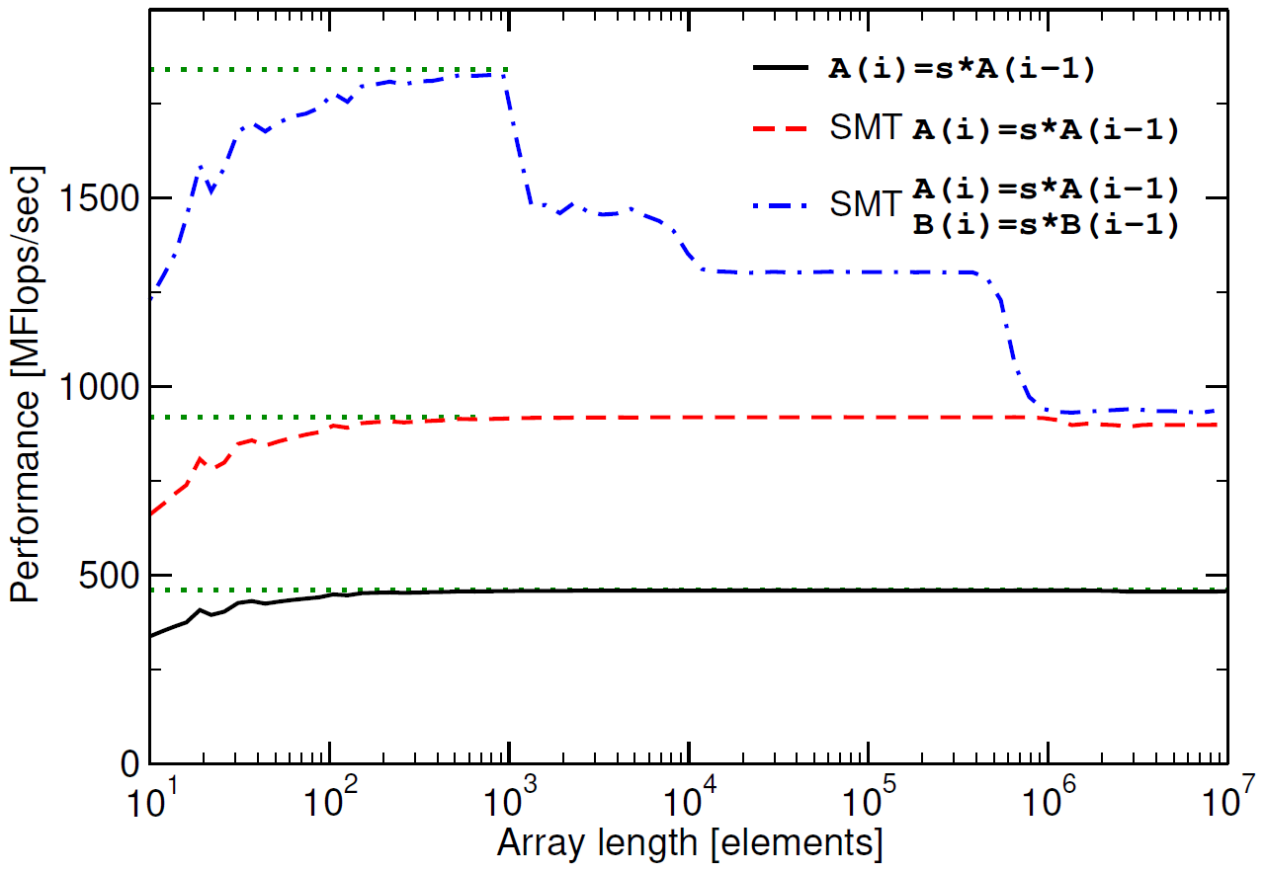


The ideal workload for SMT

Simple loop-carried dependency benchmark **`A(i) = s*A(i-1)`**

- Bottleneck: MULT pipeline latency
- Performance insensitive to problem size w/o SMT
- Fill bubbles via
  - SMT
  - Multiple indep. streams

Intel Xeon “Haswell” E5-2695v3, 2.3GHz



SMT summary

- Simultaneous multi-threading (SMT) is an in-core feature
  - Run >1 instruction stream (thread/process) at a time
  - Fill each other's pipeline bubbles
  - Full register set available per SMT thread
  - Almost all other resources are shared
  - → may improve instruction throughput in the core

- If the bottleneck is not core execution, SMT will be of limited use

- Implementations
  - Intel Xeon (up to Cascade Lake): 2-way
  - (Intel Xeon Phi Knights Landing: 4-way) ← dead!
  - AMD Zen/Zen2: 2-way
  - IBM POWER: up to 8-way
  - Marvell ThunderX2: 4-way